# Tuning Approximate Boyer-Moore for Gene Sequences*

Petri Kalsi, Leena Salmela, and Jorma Tarhio

Helsinki University of Technology
{pkalsi, lsalmela, tarhio}@cs.hut.fi

**Abstract.** Recently a new variation of approximate Boyer-Moore string matching was presented for the $k$-mismatch problem. This variation was developed for gene sequences. We further tuned this algorithm gaining speedups in both preprocessing and search times. Our preprocessing has lower time complexity than the previous algorithm and our experiments show that our algorithm is over 30% faster than the previous one. We also present two variations of the algorithm for the $k$-difference problem.

## 1 Introduction

We consider two variations of approximate string matching, the $k$-mismatch problem and the $k$-difference problem. In both of the problems, we have a pattern $p = p_0, \ldots, p_{m-1}$ of $m$ characters drawn from an alphabet $\Sigma$ of size $\sigma$ and a text $t = t_0, \ldots, t_{n-1}$ of $n$ characters over the same alphabet. We need to find all such substrings of the text that the distance between the substring and the pattern is at most $k$. In the $k$-difference problem the distance between two strings is the standard edit distance where mismatches, deletions and insertions are allowed. The $k$-mismatch problem is a more restricted one using the Hamming distance where only mismatches are allowed.

Several algorithms [12] for both variations of approximate string matching have been presented. Many of the algorithms have been developed with text data in mind and these algorithms do not necessarily work well with a small alphabet. Recently developing algorithms for small alphabets has attracted attention as approximate searching of large volumes of gene sequences has become common. One example of such a biological problem is the gene sequence acquisition problem in which a collection of gene sequences and a primer is given and we need to extract all those sequences that contain the primer with at most $k$ mismatches.

The approximate Boyer-Moore (ABM) algorithm [14] is an adaptation of the Boyer-Moore-Horspool algorithm [8] to approximate matching. ABM performs well on moderately large alphabets and low error levels. ABM was originally not designed for small alphabets and in fact it performs rather poorly on them. Liu et al. [9] tuned the $k$-mismatch version of ABM for smaller alphabets. Their algorithm, called FAAST, has a stronger shift function which makes it faster than ABM.

* Work supported by Academy of Finland.

In this paper we introduce improvements to the FAAST algorithm gaining considerable speedups in both preprocessing and search times. The preprocessing we present is simpler having a lower time complexity than that of FAAST. While the FAAST algorithm can only handle the $k$-mismatch problem, we show that with the simpler preprocessing the algorithm can be modified to also handle the $k$-difference problem.

## 2 Previous Work

So far many algorithms have been developed based on Boyer-Moore string matching [5] for the $k$-mismatch problem. Here we consider mainly ABM [14] and FAAST [9], but two other variations developed by Baeza-Yates & Gonnet [3] and El-Mabrouk & Crochemore [6] are worth mentioning. The shift function of the Baeza-Yates-Gonnet algorithm is based on the triangular inequality, whereas the El-Mabrouk-Crochemore algorithm applies the Shift-Add approach [2]. Three [6,9,14] of these four algorithms have been shown to be sublinear on the average. E.g. the average case complexity of ABM (without preprocessing) is $O(nk(1/(m-k)+k/\sigma))$.

Typically algorithms of Boyer-Moore type have two phases: preprocessing of the pattern and searching of its occurrences in the text. ABM uses the bad character rule for shifting and is thus a direct generalization of the Boyer-Moore-Horspool algorithm [8]. Instead of stopping at the first mismatch in the matching loop, the control stops at the $k{+}1^{\text{st}}$ mismatch or when an occurrence of the whole pattern is found. The shift is calculated considering $k+1$ characters currently aligned with the end of the pattern. The shift is the minimum of the precomputed shifts for those $k{+}1$ characters. After shifting, at least one of the these characters will be aligned correctly with the pattern.

FAAST is an improved variation of ABM for small alphabets using a variation of the Four-Russians technique [1,10,15] to speed up the search. Instead of minimizing $k+1$ shifts during search, it uses a precomputed shift table for a $(k+x)$-gram aligned with the end of the pattern, where $x \geq 1$ is a parameter of the algorithm. The shift table is calculated so that after the shift at least $x$ characters are aligned correctly. It is obvious that this stronger requirement leads to longer shifts in most situations, when $x > 1$ holds, and the shift is never shorter than the shift of ABM. Note that for $x = 1$ the length of shift is the same for both the algorithms, but the shift is minimized during preprocessing only in FAAST. So the algorithms are different even for $x = 1$. The optimal value of $x$ for maximum searching speed depends on other problem parameters and the computing platform. However, an increment of $x$ makes the preprocessing time grow. FAAST presents a clear improvement on solving the $k$-mismatch problem on DNA data as compared to the ABM algorithm. The preprocessing phase of FAAST is advanced because it includes the minimization step of ABM. The preprocessing time of FAAST is $O((k+x)((m-k)\sigma^{k+x}+m))$.

## 3 Algorithm for the $k$-Mismatch Problem

Our aim is to develop a faster algorithm for DNA data based on FAAST which uses a $(k+x)$-gram for shifting. We make two major changes to FAAST. We implement a simpler and faster preprocessing phase based on dynamic programming. FAAST counts the number of mismatches in the $(k+x)$-gram aligned with the end of the pattern during the searching phase. Our approach makes it possible to compute this number during preprocessing, which improves the searching speed.

The preprocessing phase computes the Hamming distance between an arbitrary $(k+x)$-gram and each $(k+x)$-gram of the pattern using dynamic programming. The first row and column of the dynamic programming table are initialized to 0, and the rest of the table can be filled with a simple iteration:

$$D[i,j] = D[i-1,j-1] + \alpha \quad \text{where } \alpha = \begin{cases} 0 & \text{if } t_{i-1} = p_{j-1}, \\ 1 & \text{otherwise} \end{cases}$$

Note that all $(k+x)$-grams of the pattern are handled in the same table. As an example, let us consider a situation where a pattern $p =$ "ggcaa" has been aligned with the text string "gcata", and $k = x = 2$ holds. The reference $(k+x)$-gram is now "cata", and the corresponding Hamming distance table of size $(k+x+1) \times (m+1)$, calculated during preprocessing, is shown in Fig. 1. First of all, we see that the last cell $D[k+x,m] = 3 > k$, and therefore it is not possible to find a match at this position, as already the suffix of the aligned text string contains too many mismatches. Otherwise, we would have to check for a match by examining the amount of mismatches in the beginning of the aligned string.

We will also look at the bottom row of the table, and find the rightmost cell $D[k+x,j]$ with a value $h \leq k$, except for the last cell $D[k+x,m]$. This is the next possible candidate for aligning the pattern with the text with less than $k$ mismatches and the correct shift is equal to $m-j$. In our example, the cell $D[k+x,2] = 2$, and we would shift the pattern by $5-2=3$ positions to get the next alignment.

| | | $D$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | | g | g | c | a | a |
| $i \backslash j$ | | 0 | 1 | 2 | 3 | 4 | 5 |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| c | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| a | 2 | 0 | 1 | 2 | 2 | 0 | 1 |
| t | 3 | 0 | 1 | 2 | 3 | 3 | 1 |
| a | 4 | 0 | 1 | **2** | 3 | 3 | **3** |

**Fig. 1.** The Hamming distance table $D$ of size $(k+x+1) \times (m+1)$ for $k$-mismatch problem ($k=2$, $x=2$). The pattern is "ggcaa" and the reference $(k+x)$-gram is "cata".

We do not need the whole table to obtain this information, so we just store the calculated Hamming distance for each generated $(k+x)$-gram in a table $M$ which is indexed by a number obtained by transforming the $(k+x)$-gram to an integer. The precalculated shifts are stored in a table $D_{kx}$. During the searching phase we convert the last $(k+x)$-gram of the aligned text string into an index $y$ to the tables, and check for an occurrence if $M[y] \leq k$. Note that if the text is not pure DNA data, we need to check the whole aligned text string against the pattern as there might be some indeterminate characters. Finally, we shift the pattern according to $D_{kx}[y]$.

We can improve the preprocessing time by applying the technique used previously by Fredriksson and Navarro [7] for approximate matching and Navarro et al. [13] for indexed approximate matching. If the $(k+x)$-grams are generated in the lexicographical order, the dynamic programming table differs only by the last few rows. Therefore we can speed up the preprocessing if we only recalculate the last rows of the table at each step, starting from the first changed character.

This can be implemented by traversing the trie built of all $(k+x)$-grams in depth first order. Nodes at the $i^{\text{th}}$ level of the trie correspond to strings of length $i$. Thus there are $\sigma^i$ nodes on level $i$ and the total number of nodes in the trie is

$$\sum_{i=1}^{k+x} \sigma^i = \sigma \frac{\sigma^{k+x} - 1}{\sigma - 1} = O(\sigma^{k+x}).$$

If we have the dynamic programming table for a node in the trie, the tables for the children nodes can be obtained by calculating one more row to the dynamic programming table taking $O(m)$ time per child so calculating the dynamic programming tables for all nodes in the trie takes $O(\sigma^{k+x}m)$ time. At the leaf nodes we have the dynamic programming table for the corresponding $(k+x)$-gram and we need to figure out the number of mismatches entered to table $M$ and the shift value entered to table $D_{kx}$ which takes $O(m)$ time. The extra calculation needed at leaf nodes is thus $O(\sigma^{k+x}m)$ because there are $\sigma^{k+x}$ leaf nodes. Therefore the time complexity of the preprocessing phase is $O(2\sigma^{k+x}m) = O(\sigma^{k+x}m)$. Note that if we implement the traversing of the trie by recursion, we actually do not need to explicitly build the trie.

We call this algorithm for the $k$-mismatch problem Algorithm 1. The shift behaviors of Algorithm 1 and FAAST are exactly the same. In FAAST the number of mismatches in the last $(k+x)$-gram of an alignment is computed during the searching phase whereas in Algorithm 1 this is fetched from a table. However, we still need to read the $(k+x)$-gram and thus the time complexity of the search phase of Algorithm 1 is the same as in FAAST.

*Implementation note.* For maximum performance it is crucial how the value of a $(k+x)$-gram is computed during searching. We mapped the ASCII values of DNA characters to integers $\{0, 1, 2, 3\}$ and used a shift-or loop to construct a bit representation of a $(k+x)$-gram.

## 4 Algorithms for the $k$-Difference Problem

Algorithm 1 can be easily modified to solve the $k$-difference problem. We initialize the dynamic programming table as in the $k$-mismatch case, but now we apply the traditional equations for the $k$-difference problem

$$D[i,j] = \min \left\{ \begin{array}{l} D[i-1,j-1] + \alpha, \\ D[i-1,j] + 1, \\ D[i,j-1] + 1 \end{array} \right\} \quad \text{where } \alpha = \begin{cases} 0 & \text{if } t_{i-1} = p_{j-1}, \\ 1 & \text{otherwise} \end{cases}$$

As before we construct the $(k+x+1) \times (m+1)$ table during preprocessing for each possible text string, and obtain the tables $M[y]$ and $D_{kx}[y]$ by checking the bottom row of the constructed table. The searching phase starts by aligning the pattern against the text prefix ending at position $m-k-1$. When examining an alignment ending at position $s$ all matches ending before that position have been reported. At each alignment we have to construct a full $(m+k+1) \times (m+1)$ edit distance table $D$ with the currently aligned text $t_{s-(m+k)+1} \ldots t_s$ against the pattern, if $M[t_{s-(k+x)+1} \ldots t_s] \leq k$. A match will be reported, if $D[m+k, m] \leq k$. After this operation we will shift the pattern according to $D_{kx}$. In order to observe correctly an occurrence of the pattern in the beginning of the text, we assume that $t_{-k}, ..., t_{-1}$ hold a character not in the pattern. The modification of Algorithm 1 for the $k$-difference problem is called Algorithm 2.

Example tables for the $k$-difference problem are shown in Fig. 2, using a pattern "ggcaa", a text string "aggcata" and parameters $k = x = 2$. We can see from the first table that $D_{kx}[\text{"cata"}] = 5-4 = 1$ and $M[\text{"cata"}] = D_0[k+x, m] = 1$. Therefore, we would construct a table $D$, and find that $D[m+k, m] = 1 \leq k$, and report a match at position $s$. We would continue the search by shifting the pattern by 1.

In the $k$-mismatch problem we did not need to reread the last $k+x$ characters from the text alignment when checking for an occurrence. Instead we had stored the number of mismatches in the table $M$ and we could extend the match based on that information. For the $k$-difference problem the situation is not quite as simple because we need to compute the dynamic programming table to check for an occurrence. The problem with Algorithm 2 is that when checking for an occurrence the aligned text is read forward while during the preprocessing phase we have generated the dynamic programming table for the last characters of the pattern. In order to use that information and avoid rereading the last $k+x$ characters we need to invert the calculation of the dynamic programming table so that we start building the table from the end of the pattern and the text string.

First we will explain how the inverted table is built and then show how that information is used to speed up the checking of an occurrence. The initialization of the inverted table is different, as we set $D[0,j] = j$ and $D[i,0] = i$ for $i \in [0, k+x], j \in [0, m]$, instead of 0. We have to read the pattern and text in reverse, and therefore we get a new condition for $\alpha$:

$$\alpha = \begin{cases} 0 & \text{if } t_{k+x-i} = p_{m-j}, \\ 1 & \text{otherwise} \end{cases}$$

$D_0$

| | | g | g | c | a | a |
|---|---|---|---|---|---|---|
| $i\backslash j$ | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| c 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| a 2 | 0 | 1 | 2 | 1 | 0 | 1 |
| t 3 | 0 | 1 | 2 | 2 | 1 | 1 |
| a 4 | 0 | 1 | 2 | 3 | **2** | **1** |

$D$

| | | g | g | c | a | a |
|---|---|---|---|---|---|---|
| $i\backslash j$ | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| a 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| g 2 | 0 | 0 | 1 | 2 | 1 | 1 |
| g 3 | 0 | 0 | 0 | 1 | 2 | 2 |
| c 4 | 0 | 1 | 1 | 0 | 1 | 2 |
| a 5 | 0 | 1 | 2 | 1 | 0 | 1 |
| t 6 | 0 | 1 | 2 | 2 | 1 | 1 |
| a 7 | 0 | 1 | 2 | 3 | 2 | **1** |

$D_{inv}$

| | | a | a | c | g | g |
|---|---|---|---|---|---|---|
| $i\backslash j$ | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| a 1 | 1 | 0 | 1 | 2 | 3 | 4 |
| t 2 | 2 | 1 | 1 | 2 | 3 | 4 |
| a 3 | 3 | 2 | 1 | 2 | 3 | 4 |
| c 4 | 4 | 3 | 2 | 1 | 2 | 3 |
| g 5 | 5 | 4 | 3 | 2 | 1 | **2** |
| g 6 | 6 | 5 | 4 | 3 | 2 | **1** |
| a 7 | 7 | 6 | 5 | 4 | 3 | **2** |

**Fig. 2.** Normal and inverted edit distance tables for $k$-difference problem ($k = 2$, $x = 2$) with the pattern "ggcaa" and the aligned text "aggcata". Sizes of the tables are $(k + x + 1) \times (m + 1)$ for $D_0$ and $(m + k + 1) \times (m + 1)$ for $D$ and $D_{inv}$.

This inverted table gives equivalent results when it comes to calculating the actual edit distance between the pattern and the aligned text string, but we still need to obtain the tables $D_{kx}$ and $M$ from a normal table. When the inverted edit distance table has been finished, we have to search for a match at the last column. To be exact, we need to check $2k + 1$ different cells of the table for a possible match, because the match can contain up to $k$ insert or delete operations, and the match length can therefore vary. All possible matches that end in the character $t_s$ will be found in the last cells of the last column of the inverted table. We can either report the first match with less than $k+1$ differences, or search for the match with the minimum differences. The current alignment $t_{s-(m+i)+1} \dots t_s$ matches the pattern $p_0 \dots p_{m-1}$ with less than $k + 1$ differences, if

$$D_{inv}[m + i, m] \leq k, i \in -k \dots k$$

If we have an edit distance table calculated for the text suffix $t_{s-(k+x)+1} \dots t_s$, we can check for a complete occurrence by filling the rest of the table rows from $t_{s-(k+x)}$ down to $t_{s-(m+k-1)}$. We can therefore store the last row of the inverted table $D_{inv}[k + x, j],\ j \in [0, m]$ for each $(k + x)$-gram during the preprocessing phase. This row can then be used to fill up the rest of the table by dynamic programming during the search phase, when the aligned text needs to be checked for an occurrence, and we do not need to run the dynamic programming for the whole table every time. We modify Algorithm 2 to use the inverted table during the search phase, and we also store the last row of the inverted tables generated during the preprocessing phase. The new algorithm is called Algorithm 3, and its pseudo code is given in Fig. 3. For simplicity, the preprocessing part of the pseudo code does not use the optimization of generating the $(k + x)$-grams in lexicographic order and recalculating the dynamic programming table only for those rows that have changed.

The preprocessing phase of Algorithm 2 has the same time complexity as that of Algorithm 1. In Algorithm 3, we need to calculate both the original dynamic

**preprocess** ($p$, $m$, $k$, $x$)

1. `for` $(i \in 0 \dots k+x)$
2. $\quad D[i,0] \leftarrow 0$
3. $\quad D_{inv}[i,0] \leftarrow i$
4. `for` $(j \in 0 \dots m)$
5. $\quad D[0,j] \leftarrow 0$
6. $\quad D_{inv}[0,j] \leftarrow j$
7. `for` $(t = t_0 \dots t_{k+x-1} \in \Sigma^{k+x})$
8. $\quad$ `for` $(i \in 1 \dots k+x,\ j \in 1 \dots m)$
9. $\quad\quad D[i,j] \leftarrow \min \left\{ \begin{matrix} D[i-1,j-1]+\alpha, \\ D[i-1,j]+1, \\ D[i,j-1]+1 \end{matrix} \right\}, \alpha = \left\{ \begin{matrix} 0 \text{ if } t_{i-1} = p_{j-1}, \\ 1 \text{ otherwise} \end{matrix} \right.$
10. $\quad\quad D_{inv}[i,j] \leftarrow \min \left\{ \begin{matrix} D_{inv}[i-1,j-1]+\alpha, \\ D_{inv}[i-1,j]+1, \\ D_{inv}[i,j-1]+1 \end{matrix} \right\}, \alpha = \left\{ \begin{matrix} 0 \text{ if } t_{k+x-i} = p_{m-j}, \\ 1 \text{ otherwise} \end{matrix} \right.$
11. $\quad M[t] \leftarrow D[k+x,m]$
12. $\quad lastRow[t] \leftarrow D_{inv}[k+x]$
13. $\quad$ `for` $(j \in [m-1,0])$
14. $\quad\quad$ `if` $D[k+x,j] < k$
15. $\quad\quad\quad D_{kx}[t] \leftarrow m-j$
16. $\quad\quad\quad$ `break`

**search** ($t$, $n$, $k$, $x$)

1. `for` $(i \in 0 \dots m+k)$
2. $\quad D_{inv}[i,0] \leftarrow i$
3. `for` $(j \in 0 \dots m)$
4. $\quad D_{inv}[0,j] \leftarrow j$
5. $s \leftarrow m-k-1$
6. `while` $(s < n)$
7. $\quad$ `if` $(M[t_{s-(k+x)+1} \dots t_s] \leq k)$ `/* possible occurrence */`
8. $\quad\quad D_{inv}[k+x] \leftarrow lastRow[t_{s-(k+x)+1} \dots t_s]$
9. $\quad\quad$ `for` $(j \in 1 \dots m,\ i \in k+x+1 \dots m+k)$
10. $\quad\quad\quad D_{inv}[i,j] \leftarrow \min \left\{ \begin{matrix} D_{inv}[i-1,j-1]+\alpha, \\ D_{inv}[i-1,j]+1, \\ D_{inv}[i,j-1]+1 \end{matrix} \right\}, \alpha = \left\{ \begin{matrix} 0 \text{ if } t_{s-i+1} = p_{m-j}, \\ 1 \text{ otherwise} \end{matrix} \right.$
11. $\quad\quad$ `if` $(D_{inv}[m+i,m] \leq k, i \in -k \dots k)$
12. $\quad\quad\quad$ `Report match at` $t_{s-(m+i)+1} \dots t_s$ `with` $D_{inv}[m+i,m]$ `differences`
13. $\quad s \leftarrow s + D_{kx}[t_{s-(k+x)+1} \dots t_s]$

**Fig. 3.** Algorithm 3 preprocessing and search phases.

**Table 1.** Search times in seconds for $k$-mismatch, using best observed $x$-values. Preprocessing times are in parentheses.

| | $k=1$ | | | $k=2$ | | |
|---|---|---|---|---|---|---|
| $m$ | ABM | FAAST | Alg.1 | ABM | FAAST | Alg.1 |
| 15 | 7.28 (0.04) | 1.17 (0.48) | 0.64 (0.03) | 15.65 (0.04) | 2.17 (1.76) | 1.21 (0.16) |
| 20 | 7.28 (0.07) | 0.92 (0.65) | 0.54 (0.03) | 15.59 (0.08) | 1.68 (2.58) | 0.98 (0.14) |
| 25 | 7.24 (0.09) | 0.78 (0.87) | 0.44 (0.04) | 15.63 (0.09) | 1.47 (3.13) | 0.81 (0.22) |
| 30 | 7.22 (0.15) | 0.68 (0.98) | 0.40 (0.06) | 15.71 (0.10) | 1.30 (3.70) | 0.69 (0.20) |
| 35 | 7.34 (0.18) | 0.60 (1.22) | 0.36 (0.05) | 15.65 (0.16) | 1.22 (4.16) | 0.53 (0.24) |
| 40 | 7.31 (0.24) | 0.53 (1.42) | 0.33 (0.05) | 15.69 (0.19) | 1.11 (4.73) | 0.54 (0.27) |

programming table and the reversed one. Because a $(k+x)$-gram is read in opposite directions when calculating these two tables we have to enumerate the $(k+x)$-grams twice. However the asymptotic time complexity remains the same.

The shifts in the searching phase of Algorithm 2 and 3 are somewhat shorter than in Algorithm 1 because the probability of two strings matching with distance less than $k$ is higher when using the standard edit distance than when using the Hamming distance.

## 5 Experimental Results

Tests were run on an Intel Pentium D 2.8 GHz dual core CPU with 1 gigabyte of memory. This processor contains 24+32 kilobytes of L1 cache, and 1024 kilobytes of L2 cache. Algorithms were implemented in C and compiled with `gcc 4.0.2`, using optimization level `-O2` as 32-bit binaries. All the algorithms were run three times for the same patterns, and the listed search and preprocessing times are the average values observed between all runs. For comparison in the $k$-mismatch case we use the original ABM algorithm and our implementation of FAAST. The Myers algorithm [11], the algorithm by Baeza-Yates and Perleberg (BYP) [4] and a version of ABM are used for the $k$-difference problem. All the results are shown with the $x$-value gaining the fastest searching speed in FAAST and our new algorithms if otherwise is not stated. The best $x$-value is generally the same for our algorithms and for FAAST. The other algorithms do not utilize the $x$-value.

The searched text is a two megabytes long sequence of the fruit fly genome. The test patterns have been extracted randomly from the text. Each pattern set consists of 200 different patterns of the same length, and they are searched sequentially.

Table 1 shows the search times for the original ABM, FAAST and Algorithm 1 in the $k$-mismatch problem. Algorithm 1 is generally 30–50% faster than FAAST in the $k$-mismatch case for $k \in [1, 3]$. Also, the preprocessing phase of Algorithm 1 is 10 to 30 times faster than that of FAAST.

Experimental results for the $k$-difference problem are shown in Table 2, and Fig. 4 further illustrates the results with $k = 2$. In the $k$-difference problem,

**Table 2.** Search times in seconds for $k$-difference, using best observed $x$-values.

| | $k=1$ | | | | | $k=2$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $m$ | ABM | Myers | BYP | Alg.2 | Alg.3 | ABM | Myers | BYP | Alg.2 | Alg.3 |
| 15 | 8.82 | 7.35 | 2.85 | 1.98 | 1.65 | 38.58 | 7.33 | 6.90 | 6.70 | 5.04 |
| 20 | 8.27 | 7.41 | 2.74 | 1.63 | 1.44 | 27.24 | 7.36 | 4.50 | 5.75 | 4.53 |
| 25 | 7.99 | 7.34 | 2.69 | 1.41 | 1.34 | 19.49 | 7.37 | 3.79 | 5.58 | 4.09 |
| 30 | 8.07 | 7.37 | 2.67 | 1.32 | 1.15 | 14.80 | 7.37 | 3.89 | 5.61 | 4.03 |
| 35 | 8.07 | - | 2.62 | 1.29 | 1.13 | 12.48 | - | 3.73 | 5.77 | 4.00 |
| 40 | 7.99 | - | 2.63 | 1.23 | 1.05 | 11.08 | - | 3.94 | 5.95 | 4.04 |

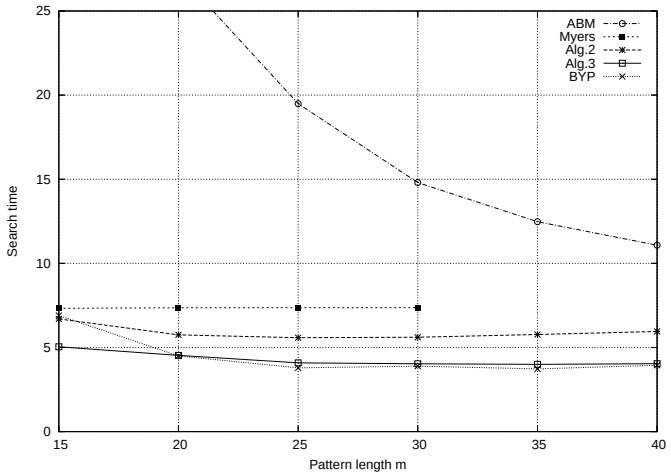


**Fig. 4.** Search times for $k$-difference with $k = 2$.

our new algorithms compare well against the Myers, BYP and ABM algorithms. Algorithms 2 and 3 are up to 50% faster than the BYP algorithm with $k = 1$, as it is shown in Table 2. For $k = 2$ Algorithm 3 is faster than BYP for short patterns but the BYP algorithm takes the lead for longer patterns. If we allow more differences, Myers is the fastest for short patterns and BYP for long ones. The basic version of the Myers algorithm is limited by the 32-bit word size, and it cannot handle patterns where $m > 32$. The modifications in Algorithm 3 decrease search time by 20-30%, when compared to Algorithm 2.

We also ran some preliminary tests to compare Algorithm 3 and the algorithm by Fredriksson and Navarro [7]. We used the version of their algorithm that reads the window backwards. In these tests Algorithm 3 was faster for pattern lengths up to 18 when $k = 1$ and up to pattern length 15 when $k = 2$. For longer patterns the algorithm by Fredriksson and Navarro was faster.

The effect of increasing the precalculated edit distance table size, and thus increasing preprocessing time with a large $x$-value is shown in Table 3. With small

**Table 3.** Preprocessing times and search times for $k$-difference, with different $x$-values ($k = 1$, $m = 20$)

| | Preprocessing | | Search | |
|---|---|---|---|---|
| $x$ | Alg. 2 | Alg. 3 | Alg.2 | Alg.3 |
| 1 | <0.01 | <0.01 | 977.30 | 724.61 |
| 2 | 0.01 | 0.01 | 213.43 | 144.53 |
| 3 | 0.02 | 0.05 | 45.57 | 28.92 |
| 4 | 0.10 | 0.18 | 11.64 | 7.08 |
| 5 | 0.37 | 0.71 | 3.94 | 2.44 |
| 6 | 1.59 | 2.76 | 1.84 | 1.44 |
| 7 | 6.38 | 11.35 | 1.63 | 1.51 |
| 8 | 25.27 | 46.50 | 3.06 | 2.94 |
| 9 | 101.09 | 188.38 | 4.03 | 4.06 |

values of $x$, the search time decreases as the amount of preprocessing increases, but after a certain limit increasing the $x$-value will begin to slow down the search. For these pattern lengths and $k$-values the optimal $x$-value was typically 4 for the $k$-mismatch problem and 6 for the $k$-difference problem.

In the implementation of Algorithm 2, preprocessing is optimized by generating the $(k + x)$-grams in lexicographic order and recalculating the dynamic programming table only for those characters that differ form the previous $(k+x)$-gram while Algorithm 3 needs to do this recursion twice, once to generate the normal dynamic programming table and once to calculate the reversed one. Thus the preprocessing times in Table 3 are longer for Algorithm 3 than for Algorithm 2.

## 6 Concluding Remarks

We have presented improved variations of the approximate Boyer-Moore algorithm for gene sequences for both the $k$-mismatch problem and the $k$-difference problem.

This in ongoing work. Next we will try to apply bit-parallelism for the preprocessing phase. We are working also on an alphabet reduction. We developed a variation of Algorithm 1, where the DNA alphabet was mapped to the binary alphabet. This version was only a bit slower than the original version. However, for short DNA texts the total time (preprocessing + searching) was the best with the alphabet reduction. The alphabet reduction also extends the applicability of our precomputed shift to larger alphabets.

**Acknowledgments.** We thank Janne Auvinen for implementing a part of the algorithms.

## References


1. Arlazarova, V., Dinic, E., Kronrod, M., Faradzev, I.: On economic construction of the transitive closure of a directed graph. Doklady Academi Nauk SSSR **194** (1970) 487–488 (in Russian). English translation in Soviet Mathematics Doklady **11** (1975) 1209–1210
2. Baeza-Yates, R., Gonnet, G.: A new approach to text searching. Communications of the ACM **35**(10) (1992) 74–82
3. Baeza-Yates, R., Gonnet, G.: Fast string matching with mismatches. Information and Computation **108**(2) (1994) 187–199
4. Baeza-Yates, R.A., Perleberg, C.H.: Fast and practical approximate string matching. Information Processing Letters **59**(1) (1996) 21–27
5. Boyer, R., Moore, J.: A fast string searching algorithm. Communications of the ACM **10**(20) (1977) 762–772
6. El-Mabrouk, N., Crochemore, M.: Boyer-Moore strategy to efficient approximate string matching. In: Proceedings of 7th Symposium on Combinatorial Pattern Matching. Volume 1075 of LNCS, Berlin, Springer-Verlag (1996) 24–38
7. Fredriksson, K., Navarro, G.: Average-optimal single and multiple approximate string matching. ACM Journal of Experimental Algorithmics **9** (2004) 1–47
8. Horspool, N.: Practical fast searching in strings. Software Practice & Experience **10** (1980) 501–506
9. Liu, Z., Chen, X., Borneman, J., Jiang, T.: A fast algorithm for approximate string matching on gene sequences. In: Proceedings of 16th Symposium on Combinatorial Pattern Matching. Volume 3537 of LNCS, Berlin, Springer-Verlag (2005) 79–90
10. Masek, W., Paterson, M.: A faster algorithm for computing string edit distances. Journal of Computer and System Sciences **20** (1980) 18–31
11. Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM **46**(3) (1999) 395–415
12. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys **33**(1) (2001) 31–88
13. Navarro, G., Sutinen, E., Tanninen, J., Tarhio, J.: Indexing text with approximate $q$-grams. In: Proceedings of 11th Symposium on Combinatorial Pattern Matching. Volume 1848 of LNCS, Berlin, Springer-Verlag (2000) 350–363
14. Tarhio, J., Ukkonen, E.: Approximate Boyer-Moore string matching. SIAM Journal on Computing **22** (1993) 243–260
15. Wu, S., Manber, U., Myers, E.: A subquadratic algorithm for approximate limited expression matching. Algorithmica **15**(1) (1996) 50–67